

Stack und Heap

- Wird ein Programm ausgeführt, so benötigt es zur Laufzeit Speicher für seine Daten:
 - einfache Variablen
 - Parameter
 - Arrays
 - Objekte
 - ...
- Dieser kann zwei verschiedenen Bereichen zugeordnet werden:
 - Stack (Stapel)
 - Heap (Halde)

Der Stack (1)

- Der Stack ist ein dynamisch wachsender Speicherbereich mit zwei wesentlichen Operationen:

push Neue Daten “oben” auf dem Stack ablegen

pop Die obersten Daten vom Stack entfernen und zurückgeben

- Daten, die zuletzt auf dem Stack abgelegt wurden, werden also zuerst wieder zurückgegeben
 - LIFO-Prinzip (Last-In-First-Out)

Verwirrendes Detail: Stacks wachsen in der Regel “nach unten”, also von hohen zu niedrigen Speicheradressen. Wenn also Daten “oben” auf den Stack gelegt werden, so liegen diese an niedrigeren Adressen als die Daten “darunter”.

Der Stack (2)

- Mit jedem Methodenaufruf wird ein neuer **Stack-Frame** erzeugt
 - Parameter, die an die Methode übergeben werden, landen auf dem Stack
 - Lokale Variablen, die innerhalb der Methode erzeugt werden, landen auf dem Stack
 - Die Rücksprungsadresse zum Aufrufer (caller) der Methode landet auf dem Stack
- Beim Verlassen einer Methode wird ihr kompletter Stack-Frame abgebaut
 - Alle darauf gespeicherten Daten sind damit verloren

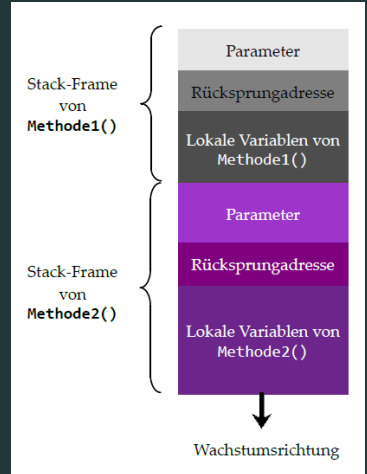


Abbildung 1: Darstellung des Stacks

- Beim Heap handelt es sich um einen Speicherbereich, auf dem Speicher für (beliebig große) Daten reserviert und wieder freigegeben werden kann.
- Im Gegensatz zum Stack erfolgt die Belegung und Freigabe nicht in einer bestimmten Reihenfolge.

Wie funktioniert das in Java?

- Stack:
 - lokale Variablen von Methoden (einschließlich Parametern)
- Heap:
 - alles, was mit **new** erzeugt wird, also insbesondere **alle Objekte** sowie zur Laufzeit erzeugte Arrays.

Beispiel:

```
void bla() {  
    Ball b = new Ball();  
    int x = b.getX();  
    // ...  
}
```

- Die Referenz **b** liegt auf dem Stack
- Das zugehörige Objekt liegt auf dem Heap
- Die Variable **x** liegt auf dem Stack
- Nach Ablauf von **bla()** existieren **b** und **x** nicht mehr

Was ist mit dem Objekt auf dem Heap?

Der Garbage-Collector

- In Java muss sich der Programmierer nicht um die Freigabe von nicht mehr benötigtem Speicher auf dem Heap kümmern.
- Der sogenannte Garbage Collector (GC) wird periodisch ausgeführt und erkennt nicht mehr benötigte Objekte
 - Der von diesen Objekten belegte Speicher wird freigegeben
- Woher weiß der GC, dass ein Objekt nicht mehr benötigt wird?
 - Der GC folgt, beginnend bei einer festgelegten “Wurzel”, allen Referenzen und markiert die so erreichbaren Objekte
 - Nach Ablauf dieser Phase gelten alle nicht markierten Objekte als nicht mehr benötigt und werden irgendwann entfernt
- Die Ausführungszeitpunkte des GC sind unspezifiziert, d. h. nicht mehr benötigte Objekte werden nicht notwendigerweise sofort gelöscht

Lebenszyklus von Objekten

1. Instanziierung (Erzeugung)

- Verwendung von **new**
- Aufruf des Konstruktors

2. Verwendung

- Aufrufen von Methoden

3. Zerstörung

- GC erkennt nicht mehr benötigte Objekte und gibt ggf. den Speicher frei